3

Mission-Critical Environments

Mission-critical is a commonly abused term. Some think it describes any architecture that they run; others believe it is a term for "systems that launch spacecraft." For the purpose of further discussion, we will equate mission-critical systems with business-critical systems. *Business-critical* is easy to define: Each business can simply choose what it believes to be vital to its operations.

Perhaps the most important issue to address from a technical perspective is to determine what aspects of a technical infrastructure are critical to the mission. Note the word is *aspects* and not *components*. This isn't solely about equipment and software; it is also about policies and procedures. Without going into painful detail, we will touch on five key aspects of mission-critical environments:

- High availability (HA)
- Monitoring
- Software management
- Overcomplication
- Optimization

To effectively manage and maintain any sizable mission-critical environment, these aspects must be mastered. Mission-critical architectures are typically managed by either a few focused teams or a few multidisciplinary teams. I prefer the latter because knowledge and standards tend to be contagious, and all five aspects are easy to master when aggregating the expertise of all individuals on a multidisciplinary team. Although it is not essential that every participant be an expert in any or all of these areas, it is essential that they be wholly competent in at least one area and always cognizant of the others.

Being mindful of the overall architecture is important. As an application developer, if you habitually ignore the monitoring systems, you are likely to make invalid assumptions resulting in decisions that negatively impact the business.

High Availability

Because high availability was the first item in the previous list, the first thing in your mind might be: "What about load balancing?" The criticality of an environment has absolutely nothing to do with its scale. Load balancing attempts to effectively combine multiple resources to handle higher loads—and therefore is completely related to scale. Throughout this book, we will continue to unlearn the misunderstood relationship between high availability and load balancing.

When discussing mission-critical systems, the first thing that comes to mind should be high availability. Without it, a simple system failure could cause a service interruption, and a service interruption isn't acceptable in a mission-critical environment. Perhaps it is the first thing that comes to mind due to the rate things seem to break in many production environments. The point of this discussion is to understand that although high availability is a necessity, it certainly won't save you from ignorance or idiocy.

High availability from a technical perspective is simply taking a single "service" and ensuring that a failure of one of its components will not result in an outage. So often high availability is considered only on the machinery level—one machine is *failover* for another. However, that is not the business goal.

The business goal is to ensure the services provided by the business are functional and accessible 100% of the time. Goals are nice, and it is always good to have a few unachievable goals in life to keep your accomplishments in perspective. Building a system that guarantees 100% uptime is an impossibility. A relatively useful but deceptive measurement that was widely popular during the dot-com era was the *n nines* measurement. Everyone wanted an architecture with *five nines availability*, which meant functioning and accessible 99.999% of the time.

Let's do a little math to see what this really means and why a healthy bit of perspective can make an unreasonable technical goal reasonable. Five nines availability means that of the (60 seconds/minute * 60 minutes/hour * 24 hours/day * 365 days/year =) 31,536,000 seconds in a year you must be up (99.999% * 31,536,000 seconds =) 31,535,684.64 seconds. This leaves an allowable (31,536,000 - 31,535,684.64 =) 315.36 seconds of unavailability. That's just slightly more than 5 minutes of downtime in an entire year.

Now, in all fairness, there are different perspectives on what it means to be *available*. Take online banking for example. It is absolutely vital that I be able to access my account online and transfer money to pay bills. However, being the night owl that I am, I constantly try to access my bank account at 3 a.m., and at least twice per month it is unavailable with a message regarding "planned maintenance." I believe that my bank has a maintenance window between 2 a.m. and 5 a.m. daily that it uses every so often. Although this may seem like cheating, most large production environments define high availability to be the lack of *unplanned* outages. So, what may be considered cheating could also be viewed as smart, responsible, and controlled. Planned maintenance windows (regardless of whether they go unused) provide an opportunity to perform proactive maintenance that reduces the risk of unexpected outages during non-maintenance windows.

Monitoring

High availability is necessary in mission-critical systems, but blind faith that the high availability solution is working and the architecture is always available (no matter how fault tolerant) is just crazy! Monitoring the architecture from top to bottom is necessary in a mission-critical system to ensure that failed pieces are caught early and dealt with swiftly before their effects propagate and cause the facing services to malfunction.

The typical approach to monitoring services is from the bottom up. Most monitoring services are managed by the operations group and as such tend to address their immediate needs and grow perpetually as needed. This methodology is in no way wrong; however, it is incomplete.

In my early years as a novice systems administrator, I was working on a client's systems after an unplanned outage and was asked why the problem wasn't caught earlier. The architecture in question was large and relatively complex (500,000 lines of custom perl web application code, 3 Oracle databases, and about 250 unique and unrelated daily jobs). The problem was that cron (the daemon responsible for dispatching daily jobs) got stuck writing to `/var/log/ocron` (its log file) and simply stopped running jobs. I won't explain in more detail, not because it is outside the scope of this book, but rather because I don't really understand *why* it malfunctioned. We had no idea that this could happen, and no bottom-up monitoring we had in place alerted us to this problem.

Where is this going? Bear with me a bit longer. I explained to the client that we monitor disk space, disk performance metrics, Oracle availability and performance, and a billion other things, but we weren't monitoring cron. He told me something extremely valuable that I have never forgotten: "I don't care if cron is running. I don't care if the disks are full or if Oracle has crashed. I don't care if the @#$% machine is on fire. All that matters is that my business is running."

I submit that top-down monitoring is as valuable as bottom-up monitoring. If your business is to sell widgets through your website, it doesn't really matter whether your machine is always on fire if you are smoothly selling widgets. Obviously, this statement is the other extreme, but you get the picture.

A solid and comprehensive monitoring infrastructure monitors systems from the bottom up (the systems and services), as well as from the top down (the business level). From this point on, we will call them *systems monitors* and *business monitors*, respectively.

Monitoring Implementations

Saying that you are going to monitor a web server is one thing; actually doing it is another. Even systems monitors are multifaceted. It is important to ensure that the machine is healthy by monitoring system load, memory-use metrics, network interface errors, disk space, and so on. On top of this, a web server is running, so you must monitor the number of hits it is receiving and the type and number of each type of response code (200 "OK," 302 "Redirect," 404 "Not Found," 500 "Internal Error," and so on). And, regardless of the metrics you get from the server itself, you should monitor the

ability to contact the service over HTTP and HTTPS to ensure that you can load pages. These are the most basic systems monitors.

Aside from checking the actual service using the protocols it speaks (in this case web-specific protocols), how do you get all the metrics you want to monitor? There isn't one single answer to this, but a standard protocol is deployed throughout the industry called *SNMP (simple network management protocol)*. Almost every commercially sold product comes instrumented with an SNMP agent. This means that by using the same protocol, you can ask every device on your network metric questions such as "How much disk space is used?" and "How many packets have you received?"

SNMP pretty much rules the monitoring landscape. Not only do most vendors implement SNMP in the architectural components they sell, all monitoring implementations (both commercial and free) support querying SNMP-capable devices. You'll notice that I said "most commercial components" and didn't mention free/open components. Sadly, many of the good open-source tools and free tools available are not instrumented with SNMP agents.

Probably the largest offender of this exclusion practice is the *Mail Transport Agent (MTA)*. Most MTAs completely ignore the fact that there are standardized SNMP mechanisms called *Management Information Bases (MIBs)*. The point here is not to complain about MTAs not implementing SNMP (although they should), but rather to illustrate that you will inevitably run into something you need to monitor on the systems level that doesn't speak SNMP. What happens then?

There are two ways to handle components that do not expose the needed information over SNMP. The first method is to fetch this metric via some unrelated network mechanism, such as SSH or HTTP, which has the tremendous advantage of simplicity. The second is to take this metric and export it over SNMP, making it both efficient and trivial to integrate into any monitoring system. Both methods require you first to develop an external means of determining the information you need (via a hand-written script or using a proprietary vendor-supplied tool).

Although the second choice may sound like the "right way," it certainly has a high overhead. SNMP is based on a complicated hierarchy of MIBs defined by a large and scattered set of documents. Without a very good MIB tool, it is difficult to decipher the information and an unbelievable pain to author a new MIB. To export the custom metrics you've determined over SNMP, you must extend or author a new MIB specifically for this and then either integrate an agent into the product or find an agent flexible enough to publish your MIB for you.

When is this worth it? That is a question that can only be answered by looking at the size of your architecture, the rate of product replacement, and the rate of change of your monitoring requirements. Remember, the monitoring system needs to monitor all these system metrics, as well as handle the business metrics. Business metrics, in my experience, are rarely exported over SNMP.

Some services are distributed in nature. Spread is one such example that cannot be effectively monitored with SNMP. The purpose of Spread is to allow separate nodes to

publish data to other nodes in a specific fashion. Ultimately, the health of a single Spread daemon is not all that important. Instead, what you really want to monitor is the service that Spread provides, and you can only do so by using the service to accomplish a simple task and reporting whether the task completed as expected.

Criteria for a Capable Monitoring System

What makes a monitoring system good? First and foremost, it is the time invested in it and winning, or at least fighting, the constant battles brought on by architecture, application, and business changes. Components are added and removed on an ongoing basis. Application changes happen. Business metrics are augmented or deprecated. If the monitoring infrastructure does not adapt hand-in-hand with these changes, it is useless and even dangerous. Monitoring things no longer of importance while failing to monitor newly introduced metrics can result in a false sense of security that acts like blinders.

The second crucial criterion is a reliable and extensible monitoring infrastructure. A plethora of commercial, free, proprietary, and open monitoring frameworks are available. Rather than do a product review, we'll look at the most basic capabilities required of a monitoring solution:

- SNMP support—This is not difficult to find and will support most of the systems monitoring requirements.
- Extensibility—The ability to plug in ad hoc monitors. This is needed for many custom systems monitors and virtually all business monitors. Suppose that a business sells widgets and decides that (based on regressing last week's data) it should sell at least 3 widgets every 15 minutes and at least 20 widgets every hour between 6 a.m. and 10 p.m. and at least 10 widgets every hour between 10 p.m. and 6 a.m. If, at any point in time, those goals aren't met, it is likely that something technical is wrong. A good monitoring system allows an operator to place arbitrary rules such as this one alongside the other systems monitors.
- Flexible notifications—The system must be able to react to failures by alerting the operator responsible for the service. Not only should an operator be notified, but the infrastructure should also support automatic escalation of failures if they are not resolved in a given time period.
- Custom reaction—Some events that occur can be rectified or further investigated by an intelligent agent. If expected website traffic falls below a reasonable lower bound, the system should notify the operator. The operator is likely to perform several basic diagnostic techniques as the first phase of troubleshooting. Instead of wasting valuable time, the monitoring system could have performed the diagnostics and placed the output in the payload of the failure notification.
- Complex scheduling—Each individual monitor should have a period that can be modified. Metrics that are expensive to evaluate and/or are slow to change can be queried less frequently than those that are cheap and/or volatile.

- Maintenance scheduling—Monitors should never be taken offline. The system should support input from an administrator that some service or set of services is expected to be down during a certain time window in the future. During these windows, the services will be monitored, but noticed failures will not be escalated.
- Event acknowledgment—When things break that do not affect the overall availability and quality of service, they often do not need to be addressed until the next business day. Disabling notifications manually is dangerous. Instead, the system should acknowledge the failure and suspend notifications for a certain period of time or until a certain fixed point in time.
- Service dependencies—Each monitor that is put in place is not an automaton. On any given web server there will be 10 or more individual system checks (connectivity, remote access, HTTP, HTTPS, time skew, disk space, system load, network metrics, and various HTTP response code rates, just to name a few). That web server is plugged into a switch, which is plugged into a load balancer, which is plugged into a firewall, and so on. If your monitoring infrastructure exists outside your architecture, there is a clear service dependency graph. There is no sense in alerting an operator that there is a time skew on a web server if the web server has crashed. Likewise, there is no sense in alerting that there is a time skew or a machine crash if there is a failure of the switch to which the machine in question is attached. A hierarchy allows one event to be a superset of another. System alerts that go to a person's pager should be clear, be concise, and articulate the problem. Defining clear service dependencies and incorporating those relationships into the notification logic is a must for a complete monitoring system.

Coping with Release Cycles

Most architectures, even the small and simple ones, are much more complicated than they first appear. For truly mission-critical applications, every piece must be thoroughly tested and retested before it is deployed in production. Even the simplest of architectures has hardware, operating systems, and server software. More complicated architectures include core switches and routers (which have software upgrade requirements), databases, load balancers, firewalls, email servers, and so on.

Managing the release cycles of external software, operating systems, and hardware is a challenge in mission-critical environments. Flawless upgrades are a testament to a good operations group.

Managing internal release cycles for all custom-built applications and the application infrastructure that powers a large website is a slightly different beast because the burden is no longer solely on the operations group. Development teams must have established practices and procedures, and, more importantly, they must follow them.

Internal Release Cycles

The typical production environment, mission-critical or not, has three vital components: development, staging, and production.

Development

Development is where things break regularly, and experiments take place. New architectural strategies are developed and tested here, as well as all application implementation.

In particularly large and well-funded organizations, research and development are split into two entities. In this scenario, things do not regularly break in development, and no experimentation takes place. Development is for the creation of new code to implement new business requirements.

The research architecture is truly a playground for implementing new ideas. If a substantial amount of experimentation takes place, splitting these architectures is important. After all, having a team of developers sitting by idly watching others clean up the mess of an experiment "gone wrong" is not a good financial investment.

Why research at all? If your business isn't technology, there is a good argument not to do any experimentation. However, staying ahead of competitors often means trying new things and adopting different ideas before they do. This applies equally to technology and business. A close relationship with vendors sometimes satisfies this, but ultimately, the people who live and breathe the business (your team) are likely to have a more successful hand in creating innovative solutions that address your needs.

Staging

Applications and services are built in development, and as a part of their construction, they are tested. Yet staging is the real infrastructure for testing. It is not testing to *see* whether it works because that was done in development. Instead, here it is testing to *make sure* that it works.

This environment should be as close to the production environment as possible (usually an exact replica) down to the hardware and software versions. Why? Complex systems are, by their very definition, complex. This means that things can and will go wrong in entirely unexpected ways.

The other big advantage that comes with an identical staging and production environment is that new releases need not be *pushed* (moved from staging to production). Because the environments are identical, when a new release has been staged, tested, and approved, the production traffic is simply *pointed* to the staging environment, and their roles simply switch.

Staging new releases of internal (and external) components provides a proving ground where true production loads can be tested. The interaction of changed pieces and the vast number of other components can be witnessed, debugged, and optimized. Often, the problems that arise in staging result in destaging and redeveloping.

The architecture must allow operations and development teams to watch things break, spiral out of control, and otherwise croak. Watching these things happen leads to understanding the cause and in turn leads to solutions.

Most architectures are forced to cope with two different types of internal releases. The first is the obvious next feature release of the application. This contains all the business requirements specified, built, tested, and integrated since the last release. The other type of internal release is the bug fix. These are incremental and necessary fixes to the current release running in production.

Bug fixes are usually staged in an environment that is much smaller than the current production environment. Because they are minor changes, the likelihood that they will cause an unexpected impact on another part of the architecture is small. The true mission-critical environments have three identical production environments: one for production, one for staging revisions, and another for staging releases.

Production

Production is where it all happens. But in reality, it is where nothing should happen from the perspective of developers and administrators. Things should be quiet, uneventful, and routine in a production environment. Money and time are invested in development environments and staging environments to ensure this peace of mind.

A Small Dose of Reality

Few businesses can afford to invest in both a complete development and a deployment environment. This is not necessarily a horrible thing. Business, like economics, is based on the principle of cost versus benefit (cost-benefit), and success relies on making good decisions based on cost-benefit information to increase return on investment. The introduction of technology into a business does not necessarily change this. This is perhaps one of the most difficult lessons for a technically oriented person to learn: The best solution technically is not always the right solution for the business.

Over the years, I have consulted for many a client who wanted to avoid the infrastructure costs of a solid development and staging environment. Ultimately, this is a decision that every business must make. Because it is impossible to say what *will* happen if you don't have an adequate staging environment, I'll place some numbers from my experience on the potential costs of not having good procedures and policies and maintaining the appropriate infrastructure to support them.

I worked on an architecture that had about a million dollars invested in hardware and software for the production environment, but the owner was only willing to invest $10,000 in the development and staging environment combined. With resources limited that way, proper staging and thorough developmental testing were impossible. Given that, about 1 in 5 pushes into production had a mild negative impact due to unexpected bugs, and about 1 in 500 pushes failed catastrophically. Before we judge this to be an ideological error, understand that all these decisions simply come down to business sense.

The mild mistakes were fixed either by reverting the bad fragments or with a second push of corrected code, and the catastrophic errors were handled by reverting to a previous known-good copy of the production code. And it turns out that the nature of these failures generally did not cost the business anything and produced only marginal unrealized profits.

A fully fledged staging and development environment could have cost an additional two or three million dollars. The cost of regular small mistakes and the rare catastrophic error were found to be less than the initial investment and maintenance of an architecture that could reduce the likelihood of such mistakes.

But, all businesses are not the same. If a bank took this approach…well, I wouldn't have an account there.

External Release Cycles

External release cycles are the art of upgrading software and hardware products deployed throughout an architecture that are not maintained internally. This typically constitutes 99% of most architectures and usually includes things such as machinery, operating systems, databases, and web server software just for starters.

External releases are typically easier to handle on an individual basis because they come as neatly wrapped packages from the vendor (even the open-source products). However, because 99% of the architecture consists of external products from different vendors, each with its own release cycle, the problem is compounded into an almost unmanageable mess.

On top of the complications of attempting to roll many unrelated external releases into one controlled release to be performed on the architecture, you have emergency releases that complicate the whole matter.

Emergency releases are releases that must be applied with extreme haste to solve an issue that could expose the architecture from a security standpoint or to resolve a sudden and acute issue (related to performance or function) that is crippling the business.

Examples of emergency releases are abundant in the real world:

- An exploit in the OpenSSL library is found, which sits under mod_ssl, which sits inside Apache to provide secure web access to your customers. Simply put, all your servers running that code are vulnerable to being compromised, and it is essential that you upgrade them as quickly as is safely possible.
- A bug is found in the version of Oracle used in your architecture. Some much-needed performance tuning was done, and the bug is manifesting itself acutely. You open a ticket with Oracle, and they quickly track down the bug and provide you with a patch to apply. That patch must be applied immediately because the problem is crippling the applications that are using Oracle.

The preceding examples are two of a countless number of real-life emergency rollouts that I have encountered.

The truth is that managing external releases is the core responsibility of the operations group. It is not a simple task, and an entire book (or series of books) could be written explaining best practices for this management feat.

The Cost of Complexity
Shackled to Large Architectures

It has long since been established that, in the world of technology, having even the smallest gadget grants certain bragging rights. However, for one reason or another, when it comes to the technology architecture, everyone likes to brag that he has bigger toys than the next guy.

I can't count the number of times I've overheard developers and engineers at conferences arguing about who has the larger and more complicated architecture. Since when did this become a goal? The point of building scalable systems is that they scale easily and *cost effectively*. One aspect of being cost effective is minimizing the required infrastructure, and another is minimizing the cost of maintaining that architecture. Two people arguing over who has the more complicated architecture sounds like two people arguing over who can burn money in a hotter furnace—I hope they both win.

Of course, architectures can be necessarily complicated. But a good architect should always be battling to *KISS (keep it simple, stupid)*. Two fundamental truths about complex architectures are as follows:

- Independent architectural components added to a system complicate it linearly. Dependent architectural components added complicate it exponentially.
- Complex systems of any type are complicated.

The first means that when one component relies on another component in the architecture, it compounds the complexity. An example of an independent component is a transparent web proxy-cache. The device sits out front and simply attempts to speed viewing of the site by caching content. It can be removed without affecting the function of the architecture. On the other hand, the complexity of a core database server is compounded because the web application depends on it.

Although the second truth may sound overly obvious, it is often ignored or overlooked. A complex system is more difficult to operate, develop, extend, troubleshoot, and simply be around. Inevitably, business requirements will change, and modifications will need to be made. Complex systems have a tendency to become more complex when they are modified to accomplish new tasks.

Looking for Speed

From a fundamental perspective, performance and scalability are orthogonal. Scalability in the realm of technical infrastructure simply means that it can grow and shrink without fundamental change. Growing, of course, means serving more traffic and/or more users, and that directly relates to the performance of the system as a whole.

If you have two load-balanced web servers that serve static HTML pages only, this is an inherently scalable architecture. More web servers can be added, and capacity can be increased without redesigning the system or changing dependent components (because no dependent components are in the system).

Even if your web servers can serve only one hit per second, the system scales. If you need to serve 10,000 hits per second, you can simply deploy 10,000 servers to solve your performance needs. This scales, but by no means scales well.

It may be obvious that one hit per second is terribly low. If you had used 500 hits per second as the performance of a single web server, you would only have needed 20 machines. Herein lies the painful relativity of the term *scales well*—serving 500 hits per second of static content is still an underachievement, and thus 20 machines is an unacceptably large number of machines for this task.

It should be clear from the preceding example that the performance of an individual component in the architecture can drastically affect how efficiently a system can scale. It is imperative that the performance of every introduced architectural component is scrutinized and judged. If a component performs the needed task but does not scale and scale well, using it will damage the scalability of the entire architecture.

Why be concerned with the performance of individual components? The only way to increase the performance of a complex system is to reduce the resource consumption of one or more of its individual components. Contrapositively, if an individual component of a complex system performs slowly, it is likely to capsize the entire architecture. It is fundamental that solid performance-tuning strategies be employed through the *entire* architecture.

Every architecture has components; every component runs software of some type. If your architecture has performance problems, it is usually obvious in which component the problems are manifesting themselves. From that point, you look at the code running on that component to find the problem. A few common scenarios contribute to the production of slow code:

- Many developers who are good at meeting design and function requirements are not as skilled in performance tuning. This needs to change.
- It is often easier to detect performance problems after the system has been built and is in use.
- People believe that performance can be increased by throwing more hardware at the problem.

Given that there is no magical solution, how does one go about writing high-performance code? This is an important question, and there is a tremendous amount of literature on the market about how to optimize code under just about every circumstance imaginable. Because this book doesn't focus on how to write high-performance code, we will jump to how to diagnose poorly performing code.

Gene Ahdmal stated that speeding up code inside a bottleneck has a larger impact on software performance than does speeding up code outside a bottleneck. This combined

with classic 90/10 principle of code (90% of execution time is spent in 10% of the code) results in a good target.

Do *not* choose the slowest architectural component or piece of code to focus on. Start with the most common execution path and evaluate its impact on the system. The thing to keep in mind is that a 50% speedup of code that executes 0.1% of the time results in an overall speedup of 0.05%, which is small. On the other hand, a 50% speedup of code that executes 5% of the time results in an overall speedup of 2.5%, which is significant.

At the end of the day, week, month, or year, there will be code that is bad. Even the best developers write bad code at times. It is important that all infrastructure code and application code be open to review and revision and that performance review and tuning is a perpetual cycle.

I honestly believe the most valuable lessons in performance tuning, whether it be on the systems level or in application development, come from building things wrong. Reading about or being shown by example how to do a task "correctly" lacks the problem-solving skills that lead to its "correctness." It also does not present in its full form the contrast between the original and the new.

By working directly on an application that had performance issues, you can work toward improvement. Realizing performance gains due to minor modifications or large refactoring has tremendous personal gratification, but there is more to it than that. The process teaches the analytical thought processes required to anticipate future problems before they manifest themselves.

Sometimes performance tuning must be "out of the box." Analysis on the microscopic level should regularly be retired to more macroscopic views. This multiresolutioned problem analysis can turn a question such as "How can I merge all these log files faster?" into "Why do I have all these log files to merge and is there a better way?" Or a question such as "How can I make this set of problematic database queries faster?" becomes "Why am I putting this information in a database?"

Changing the scope of the question allows problems to be tackled from different angles, and regular reassessment provides an opportunity to use the right tool for the job.

It's Not a One-Man Mission

If the mission is critical, trusting it to one man is folly. To meet the objective, time, effort, and money have been spent to make an architecture with no single point of failure, so having a single person managing it violates the objective. Personnel are part of the architecture.

This leaves us with more than one man and in all fairness a decent-sized multidisciplinary team. Although in large teams, it isn't cost effective to educate every person involved on every aspect of running the system—that is, after all, why it is called a multidisciplinary team—it is vital that the lines of communication be kept open so that responsibilities and knowledge are not strictly isolated.

As soon as you lose respect for any one of these key aspects you will be bitten.